***BIAX Corporation v. Intel***
**Civil Action No. 2:05-cv-184-TJW**

# EXHIBIT 1
# (PART 3)
# FIRST AMENDED COMPLAINT FOR PATENT INFRINGEMENT

example, through data chaining) to succeeding instructions which further manipulate the data. Ultimately, data is transmitted to the register file after some finite sequence of instructions completes; however, it is only the final data that is transmitted.

This can be viewed as a generalization of the case of a microcoded complex instruction as described above, and can be considered a substantially context free processor element implementation. In such an implementation, the TOLL software would be required to ensure that dependent instructions execute on the same processor element until such time as data is ultimately transmitted to the context register file. As with pipelined processor elements, this does not change the overall functionality and architecture of the TOLL software, but mainly affects the efficient scheduling of instructions among processor elements to make optimal use of each instruction cycle on all processor elements.

## DETAILED DESCRIPTION

1. Detailed Description of Software

In FIGS. **8** through **11**, the details of the TOLL software **110** of the present invention are set forth. Referring to FIG. **8**, the conventional output from a compiler is delivered to the TOLL software at the start stage **800**. The following information is contained within the conventional compiler output **800**: (a) instruction functionality, (b) resources required by the instruction, (c) locations of the resources (if possible), and (d) basic block boundaries. The TOLL software then starts with the first instruction at stage **810** and proceeds to determine "which" resources are used in stage **820** and "how" the resources are used in stage **830**. This process continues for each instruction within the instruction stream through stages **840** and **850** as was discussed in the previous section.

After the last instruction is processed, as tested in stage **840**, a table is constructed and initialized with the "free time" and "load time" for each resource. Such a table is set forth in Table 7 for the inner loop matrix multiply example and at initialization, the table contains all zeros. The initialization occurs in stage **860** and once constructed the TOLL software proceeds to start with the first basic block in stage **870**.

TABLE 7

| Resource | Load Time | Free Time |
|---|---|---|
| R0 | T0 | T0 |
| R1 | T0 | T0 |
| R2 | T0 | T0 |
| R3 | T0 | T0 |
| R4 | T0 | T0 |
| R10 | T0 | T0 |
| R11 | T0 | T0 |

Referring to FIG. **9**, the TOLL software continues the analysis of the instruction stream with the first instruction of the next basic block in stage **900**. As stated previously, TOLL performs a static analysis of the instruction stream. Static analysis assumes (in effect) straight line code, that is, each instruction is analyzed as it is seen in a sequential manner. In other words, static analysis assumes that a branch is never taken. For non-pipelined instruction execution, this is not a problem, as there will never be any dependencies that arise as a result of a branch. Pipelined execution is discussed subsequently (although, it can be stated that the use of pipelining will only affect the delay value of the branch instruction).

Clearly, the assumption that a branch is never taken is incorrect. However, the impact of encountering a branch in the instruction stream is straightforward. As stated previously, each instruction is characterized by the resources (or physical hardware elements) it uses. The assignment of the firing time (and in the illustrated embodiment, the logical processor number) is dependent on how the instruction stream accesses these resources. Within this particular embodiment of the TOLL software, the usage of each resource is represented, as noted above, by data structures termed the free and load times for that resource. As each instruction is analyzed in sequence, the analysis of a branch impacts these data structures in the following manner.

When all of the instructions of a basic block have been assigned firing times, the maximum firing time of the current basic block (the one the branch is a member of) is used to update all resources load and free times (to this value). When the next basic block analysis begins, the proposed firing time is then given as the last maximum value plus one. Hence, the load and free times for each of the register resources R0 through R4, R10 and R11 are set forth below in Table 8, for the example, assuming the basic block commences with a time of T16.

TABLE 8

| Resource | Load Time | Free Time |
|---|---|---|
| R0 | T15 | T15 |
| R1 | T15 | T15 |
| R2 | T15 | T15 |
| R3 | T15 | T15 |
| R4 | T15 | T15 |
| R10 | T15 | T15 |
| R11 | T15 | T15 |

Hence, the TOLL software sets a proposed firing time (PFT) in stage **910** to the maximum firing time plus one of the previous basic blocks firing times. In the context of the above example, the previous basic block's last firing time is T15, and the proposed firing time for the instructions in this basic block commence with T16.

In stage **920**, the first resource used by the first instruction, which in this case is register R0 of instruction I0, is analyzed. In stage **930**, a determination is made as to whether or not the resource is read. In the above example, for instruction I0, register R0 is not read but is written and, therefore, stage **940** is next entered to make the determination of whether or not the resource is written. In this case, instruction I0 writes into register R0 and stage **942** is entered. Stage **942** makes a determination as to whether the proposed firing time (PFT) for instruction I0 is less than or equal to the free time for the resource. In this case, referring to Table 8, the resource free time for register R0 is T15 and, therefore, the instruction proposed firing time of T16 is greater than the resource free time of T15 and the determination is "no" and stage **950** is accessed.

The analysis by the TOLL software proceeds to the next resource which in the case, for instruction I0, is register R10. This resource is both read and written by the instruction. Stage **930** is entered and a determination is made as to whether or not the instruction reads the resource. It does, so stage **932** is entered where a determination is made as to whether the current proposed firing time for the instruction (T16) is less than the resource load time (T15). It is not, so stage **940** is entered. Here a determination is made as to whether the instruction writes the resource. It does; so stage **942** is

entered. In this stage a determination is made as to whether the proposed firing time for the instruction (T16) is less than the free time for the resource (T15). It is not, and stage **950** is accessed. The analysis by the TOLL software proceeds either to the next resource (there is none for instruction I0) or to "B" (FIG. **10**) if the last resource for the instruction has been processed.

Hence, the answer to the determination at stage **950** is affirmative and the analysis then proceeds to FIG. **10**. In FIG. **10**, the resource free and load times will be set. At stage **1000**, the first resource for instruction I0 is register R0. The first determination in stage **1010** is whether or not the instruction reads the resource. As before, register R0 in instruction I0 is not read but written and the answer to this determination is "no" in which case the analysis then proceeds to stage **1020**. In stage **1020**, the answer to the determination as to whether or not the resource is written is "yes" and the analysis proceeds to stage **1022**. Stage **1022** makes the determination as to whether or not the proposed firing time for the instruction is greater than the resource load time. In the example, the proposed firing time is T16 and, with reference back to Table 8, the firing time T16 is greater than the load time T15 for register R0. Hence, the response to this determination is "yes" and stage **1024** is entered. In stage **1024**, the resource load time is set equal to the instruction's proposed firing time and the table of resources (Table 8) is updated to reflect that change. Likewise, stage **1026** is entered and the resource free-time is updated and set equal to the instruction's proposed firing time plus one or T17 (T16 plus one).

Stage **1030** is then entered and a determination made as to whether there are any further resources used by this instruction. There is one, register R10, and the analysis processes this resource. The next resource is acquired at stage **1070**. Stage **1010** is then entered where a determination is made as to whether or not the resource is read by the instruction. It is and so stage **1012** is entered where a determination is made as to whether the current proposed firing time (T16) is greater than the resource's free-time (T15). It is, and therefore stage **1014** is entered where the resource's free-time is updated to reflect the use of this resource by this instruction. The method next checks at stage **1020** whether the resource is written by the instruction. It is, and so stage **1022** is entered where a determination is made as to whether or not the current proposed firing time (T16) is greater than the load time of the resource (T15). It is, so stage **1024** is entered. In this stage, the resource's load-time is updated to reflect the firing time of the instruction, that is, the load-time is set to T16. Stage **1026** is then entered where the resource's free-time is updated to reflect the execution of the instruction, that is, the free-time is set to T17. Stage **1030** is then entered where a determination is made as to whether or not this is the last resource used by the instruction. It is, and therefore, stage **1040** is entered. The instruction firing time (IFT) is now set to equal the proposed firing time (PFT) of T16. Stage **1050** is then accessed which makes a determination as to whether or not this is the last instruction in the basic block, which in this case is "no"; and stage **1060** is entered to proceed to the next instruction, I1, which enters the analysis stage at "A1" of FIG. **9**.

The next instruction in the example is I1 and the identical analysis is had for instruction I1 for registers R1 and R11 as presented for instruction I0 with registers R0 and R10. In Table 9 below, a portion of the resource Table 8 is modified to reflect these changes. (Instructions I0 and I1 have been fully processed by the TOLL software.)

TABLE 9

| Resource | Load Time | Free Time |
|---|---|---|
| R0 | T16 | T17 |
| R1 | T16 | T17 |
| R10 | T16 | T17 |
| R11 | T16 | T17 |

The next instruction in the basic block example is instruction I2 which involves a read of registers R0 and R1 and a write into register R2. Hence, in stage **910** of FIG. **9**, the proposed firing time for the instruction is set to T16 (T15 plus 1). Stage **920** is then entered and the first resource in instruction I2 is register R0. The first determination made in stage **930** is "yes" and stage **932** is entered. At stage **932**, a determination is made whether the instruction's proposed firing time of T16 is less than or equal to the resource register R0 load time of T16. It is important to note that the resource load time for register R0 was updated during the analysis of register R0 for instruction I0 from time T15 to time T16. The answer to this determination in stage **932** is that the proposed firing time equals the resource load time (T16 equals T16) and stage **934** is entered. In stage **934**, the instruction proposed firing time is updated to equal the resource load time plus one or, in this case, T17 (T16 plus one). The instruction I2 proposed firing time is now updated to T17. Now stage **940** is entered and since instruction I2 does not write resource R0, the answer to the determination is "no" and stage **950** and then stage **960** are entered to process the next resource which in this case is register R1.

Stage **960** initiates the analysis to take place for register R1 and a determination is made in stage **930** whether or not the resource is read. The answer, of course, is "yes" and stage **932** is entered. This time the instruction proposed firing time is T17 and a determination is made whether or not the instruction proposed firing time of T17 is less than or equal to the resource load time for register R1 which is T16. Since the instruction proposed firing time is greater than the register load time (T17 is greater than T16), the answer to this determination is "no" and stage **940**. The register is not written by this instruction and, therefore, the analysis proceeds to stage **950**. The next resource to be processed for instruction I2, in stage **960**, is resource register R2.

The first determination of stage **930** is whether or not this resource R2 is read. It is not and hence the analysis moves to stage **940** and then to stage **942**. At this point in time the instruction I2 proposed firing time is T17 and in stage **942** a determination is made whether or not the instruction's proposed firing time of T17 is less than or equal to resources, R2 free time which in Table 8 above is T15. The answer to this determination is "no" and therefore stage **950** is entered. This is the last resource processed for this instruction and the analysis continues in FIG. **10**.

Referring to FIG. **10**, the first resource R0 for instruction I2 is analyzed. In stage **1010**, the determination is made whether or not this resource is read and the answer is "yes." Stage **1012** is then entered to make the determination whether the proposed firing time T17 of instruction I2 is greater than the resource free-time for register R0. In Table 9, the free-time for register R0 is T17 and the answer to the determination is "no" since

both are equal. Stage **1020** is then entered which also results in a "no" answer transferring the analysis to stage **1030**. Since this is not the last resource to be processed for instruction I2, stage **1070** is entered to advance the analysis to the next resource register R1. Precisely the same path through FIG. **10** occurs for register R1 as for register R0. Next, stage **1070** initiates processing of register R2. In this case, the answer to the determination at stage **1010** is "no" and stage **1020** is accessed. Since register R2 for instruction I2 is written, stage **1022** is accessed. In this case, the proposed firing time of instruction I2 is T17 and the resource load-time is T15 from Table 8. Hence, the proposed firing time is greater than the load time and stage **1024** is accessed. Stages **1024** and **1026** cause the load time and the free time for register R2 to be advanced, respectively, to T17 and T18, and the resource table is updated as shown in FIG. **10**:

TABLE 10

| Resource | Load-Time | Free-Time |
|---|---|---|
| R0 | T16 | T17 |
| R1 | T16 | T17 |
| R2 | T17 | T18 |

As this is the last resource processed, for instruction I2, the proposed firing time of T17 becomes the actual firing time (stage **1040**) and the next instruction is analyzed.

It is in this fashion that each of the instructions in the inner loop matrix multiply example are analyzed so that when fully analyzed the final resource table appears as in Table 11 below:

TABLE 11

| Resource | Load-Time | Free-Time |
|---|---|---|
| R0 | T16 | T17 |
| R1 | T16 | T17 |
| R2 | T17 | T18 |
| R3 | T18 | T19 |
| R4 | T16 | T17 |
| R10 | T16 | T17 |
| R11 | T16 | T17 |

Referring to FIG. **11**, the TOLL software, after performing the tasks set forth in FIGS. **9** and **10**, enters stage **1100**. Stage **1100** sets all resource free and load times to the maximum of those within the given basic block. For example, the maximum time set forth in Table 11 is T19 and, therefore, all free and load times are set to time T19. Stage **1110** is then entered to make the determination whether this is the last basic block to be processed. If not, stage **1120** is entered to proceed with the next basic block. If this is the last basic block, stage **1130** is entered and starts again with the first basic block in the instruction stream. The purpose of this analysis is to logically reorder the instructions within each basic block and to assign logical processor numbers to each instruction. This is summarized in Table 6 for the inner loop matrix multiply example. Stage **1140** performs the function of sorting the instruction in each basic block in ascending order using the instruction firing time (IFT) as the basis. Stage **1150** is then entered wherein the logical processor numbers (LPNs) are assigned. In making the assignment of the processor elements, the instructions of a set, that is those having the same instruction firing time (IFT), are assigned logical processor numbers on a first come, first serve basis. For example, in reference back to Table 6, the first set of instructions for firing time T16 are I0, I1, and I4. These instructions are assigned respectively to processors PE0, PE1, and PE2. Next, during time T17, the second set of instructions I2 and I5 are assigned to processors PE0 and PE1, respectively. Finally, during the final time T18, the final instruction I3 is assigned to processor PE0. It is to be expressly understood that the assignment of the processor elements could be effected using other methods and is based upon the actual architecture of the processor element and the system. As is clear, in the preferred embodiment the set of instructions are assigned to the logical processors on a first in time basis. After making the assignment, stage **1160** is entered to determine whether or not the last basic block has been processed and if not, stage **1170** brings forth the next basic block and the process is repeated until finished.

Hence, the output of the TOLL software, in this illustrated embodiment, results in the assignment of the instruction firing time (IFT) for each of the instructions as shown in FIG. **4**. As previously discussed, the instructions are reordered, based upon the natural concurrencies appearing in the instruction stream, according to the instruction firing times; and, then, individual logical processors are assigned as shown in Table 6. While the discussion above has concentrated on the inner loop matrix multiply example, the analysis set forth in FIGS. **9** through **11** can be applied to any SESE basic block (BB) to detect the natural concurrencies contained therein and then to assign the instruction firing times (IFTs) and the logical processor numbers (LPNs) for each user's program. This intelligence can then be added to the reordered instructions within the basic block. This is only done once for a given program and provides the necessary time-driven decentralized control and processor mapping information to run on the TDA system architecture of the present invention.

The purpose of the execution sets, referring to FIG. **12**, is to optimize program execution by maximizing instruction cache hits within an execution set or, in other words, to statically minimize transfers by a basic block within an execution set to a basic block in another execution set. Support of execution sets consists of three major components: data structure definitions, pre-execution time software which prepares the execution set data structures, and hardware to support the fetching and manipulation of execution sets in the process of executing the program.

The execution set data structure consists of a set of one or more basic blocks and an attached header. The header contains the following information: the address **1200** of the start of the actual instructions (this is implicit if the header has a fixed length), the length **1210** of the execution set (or the address of the end of the execution set), and zero or more addresses **1220** of potential successor (in terms of program execution) execution sets.

The software required to support execution sets manipulates the output of the post-compile processing. That processing performs dependency analysis, resource analysis, resource assignment, and individual instruction stream reordering. The formation of execution sets uses one or more algorithms for determining the probable order and frequency of execution of the basic blocks. The basic blocks are grouped accordingly. The possible algorithms are similar to the algorithms used in solving linear programming problems for least-cost routing. In the case of execution sets, cost is associated with branching. Branching between basic blocks

contained in the same execution set incurs no penalty with respect to cache operations because it is assumed that the instructions for the basic blocks of an execution set are resident in the cache in the steady state. Cost is then associated with branching between basic blocks of different execution sets, because the instructions of the basic blocks of a different execution set are assumed not to be in cache. Cache misses delay program execution while the retrieval and storage of the appropriate block from main memory to cache is made.

There are several possible algorithms which can be used to assess and assign costs under the teaching of the present invention. One algorithm is the static branch cost approach. In accordance with this method, one begins by placing basic blocks into execution sets based on block contiguity and a maximum allowable execution set size (this would be an implementation limit, such as maximum instruction cache size). The information about branching between basic blocks is known and is an output of the compiler. Using this information, the apparatus calculates the "cost" of the resulting grouping of basic blocks into execution sets based on the number of (static) branches between basic blocks in different execution sets. The apparatus can then use standard linear programming techniques to minimize this cost function, thereby obtaining the "optimal" grouping of basic blocks into execution sets. This algorithm has the advantage of ease of implementation; however, it ignores the actual dynamic branching patterns which occur during actual program execution.

Other algorithms could be used in accordance with the teachings of the present invention which provide a better estimation of actual dynamic branch patterns. One example would be the collection of actual branch data from a program execution, and the resultant regrouping of the basic blocks using a weighted assignment of branch costs based on the actual inter-block branching. Clearly, this approach is data dependent. Another approach would be to allow the programmer to specify branch probabilities, after which the weighted cost assignment would be made. This approach has the disadvantages of programmer intervention and programmer error. Still other approaches would be based using parameters, such as limiting the number of basic blocks per execution set, and applying heuristics to these parameters.

The algorithms described above are not unique to the problem of creating execution sets. However, the use of execution sets as a means of optimizing instruction cache performance is novel. Like the novelty of pre-execution time assignment of processor resources, the pre-execution time grouping of basic blocks for maximizing cache performance is not found in prior art.

The final element required to support the execution sets is the hardware. As will be discussed subsequently, this hardware includes storage to contain the current execution set starting and ending addresses and to contain the other execution set header data. The existence of execution sets and the associated header data structures are, in fact, transparent to the actual instruction fetching from cache to the processor elements. The latter depends strictly upon the individual instruction and branch addresses. The execution set hardware operates independently of instruction fetching to control the movement of instruction words from main memory to the instruction cache. This hardware is responsible for fetching basic blocks of instructions into the cache until either the entire execution set resides in cache or program execution has reached a point that a branch has occurred to a basic block outside the execution set. At this point, since the target execution set is not resident in cache, the execution set hardware begins fetching the basic blocks belonging to the target execution set.

Referring to FIG. **13**, the structure of the register set file **660** for context file zero (the structure being the same for each context file) has L+1 levels of register sets with each register set containing n+1 separate registers. For example, n could equal 31 for a total of 32 registers. Likewise, the L could equal 15 for a total of 16 levels. Note that these registers are not shared between levels; that is, each level has a set of registers which is physically distinct from the registers of each other level.

Each level of registers corresponds to that group of registers available to a subroutine executing at a particular depth relative to the main program. For example, the set of registers at level zero can be available to the main program; the set of registers at level one can be available to a first level subroutine that is called directly from the main program; the set of registers at level two can be available to any subroutine (a second level subroutine) called directly by a first level subroutine; the set of registers at level three can be available to any subroutine called directly by a second level subroutine; and so on.

As these sets of registers are independent, the maximum number of levels corresponds to the number of subroutines that can be nested before having to physically share any registers between subroutines, that is, before having to store the contents of any registers in main memory. The register sets, in their different levels, constitute a shared resource of the present invention and significantly saves system overhead during subroutine calls since only rarely do sets of registers need to be stored, for example in a stack, in memory.

Communication between different levels of subroutines takes place, in the preferred illustrated embodiment, by allowing each subroutine up to three possible levels from which to obtain a register: the current level, the previous (calling) level (if any) and the global (main program) level. The designation of which level of registers is to be accessed, that is, the level relative to the presently executing main program or subroutine, uses the static SCSM information attached to the instruction by the TOLL software. This information designates a level relative to the instruction to be processed. This can be illustrated by a subroutine call for a SINE function that takes as its argument a value representing an angular measure and returns the trigonometric SINE of that measure. The main program is set forth in Table 12; and the subroutine is set forth in Table 13.

TABLE 12

| Main Program | Purpose |
|---|---|
| LOAD X, R1 | Load X from memory into Reg R1 for parameter passing |
| CALL SINE | Subroutine Call - Returns result in Reg R2 |
| LOAD R2, R3 | Temporarily save results in Reg R3 |
| LOAD Y, R1 | Load Y from memory into Reg R1 for parameter passing |
| CALL SINE | Subroutine Call Returns result in Reg R2 |

TABLE 12-continued

| Main Program | Purpose |
|---|---|
| MULT R2, R3, R4 | Multiply Sin (x) with Sin (y) and store result in Reg R4 |
| STORE R4, Z | Store final result in memory at Z |

The SINE subrouting is set forth in Table 13:

TABLE 13

| Instruction | Subroutine | Purpose |
|---|---|---|
| I0 | Load R1(L0), R2 | Load Reg R2, level 1 with contents of Reg R1, level 0 |
| Ip-1 | (Perform SINE), R7 | Calculate SINE function and store result in Reg R7, level 1 |
| Ip | Load R7, R2(L0) | Load Reg R2, level 0 with contents of Reg R7, level 1 |

Hence, under the teachings of the present invention and with reference to FIG. **14**, instruction **I0** of the subroutine loads register **R2** of the current level (the subroutine's level or called level) with the contents of register **R1** from the previous level (the calling routine or level). Note that the subroutine has a full set of registers with which to perform the processing independent of the register set of the calling routine. Upon completion of the subroutine call, instruction Ip causes register **R7** of the current level to be stored in register **R2** of the calling routine's level (which returns the results of the SINE routine back to the calling program's register set).

As described in more detail in connection with FIG. **22**, the transfer between the levels occurs through the use of the SCSM dynamically generated information which can contain the absolute value of the current procedural level of the instruction (that is, the level of the called routine), the previous procedural level (that is, the level of the calling routine) and the context identifier. The absolute dynamic SCSM level information is generated by the LRD from the relative (static) SCSM information provided by the TOLL software. The context identifier is only used when processing a number of programs in a multi-user system. The relative SCSM information is shown in Table 13 for register **R1** (of the calling routine) as **R1(L0)** and for register **R2** as **R2(L0)**. All registers of the current level have appended an implied (00) signifying the current procedural level.

This method and structure described in connection with FIGS. **13** and **14** differ substantially from prior art approaches where physical sharing of the same registers occurs between registers of a subroutine and its calling routine. By thereby limiting the number of registers that are available for use by the subroutine, more system overhead for storing the registers in main memory is required. See, for example, the MIPS approach as set forth in "Reduced Instruction Set Computers" David A. Patterson, Communications of the ACM, January, 1985, Vol. 28, No. 1, Pgs. 8–21. In that reference, the first sixteen registers are local registers to be used solely by the subroutine, the next eight registers, registers **16** through **23**, are shared between the calling routine and the subroutine, and final eight registers, registers **24** through **31** are shared between the global (or main) program and the subroutine. Clearly, out of 32 registers that are accessible by the subroutine, only 16 are dedicated solely for use by the subroutine in the processing of its program. In the processing of complex subroutines, the limited number of registers that are dedicated solely to the subroutine may not (in general) be sufficient for the processing of the subroutine. Data shuffling (entailing the storing of intermediate data in memory) must then occur, resulting in significant overhead in the processing of the routine.

Under the teachings of the present invention, the relative transfers between the levels which are known to occur at compile time are specified by adding the requisite information to the register identifiers as shown in FIG. **4** (the SCSM data), to appropriately map the instructions between the various levels. Hence, a completely independent set of registers is available to the calling routine and to each level of subroutine. The calling routine, in addition to accessing its own complete set of registers, can also gain direct access to a higher set of registers using the aforesaid static SCSM mapping code which is added to the instruction, as previously described. There is literally no reduction in the size of the register set available to a subroutine as specifically found in prior art approaches. Furthermore, the mapping code for the SCSM information can be a field of sufficient length to access any number of desired levels. For example, in one illustrated embodiment, a calling routine can access up to seven higher levels in addition to its own registers with a field of three bits. The present invention is not to be limited to any particular number of levels nor to any particular number of registers within a level. Under the teachings of the present invention, the mapping shown in FIG. **14** is a logical mapping and not a conventional physical mapping. For example, three levels, such as the calling routine level, the called level, and the global level require three bit maps. The relative identification of the levels can be specified by a two bit word in the static SCSM, for example, the calling routine by (00), the subordinate level by (01), and the global level by (11). Thus, each user's program is analyzed and the static SCSM relative procedural level information, also designated a window code, is added to the instructions prior to the issuance of the user program to a specific LRD. Once the user is assigned to a specific LRD, the static SCSM level information is used to generate the LRD dependent and dynamic SCSM information which is added as it is needed.

2. Detailed Description of the Hardware

As shown in FIG. **6**, the TDA system **600** of the present invention is composed of memory **610**, logical resource drivers (LRD) **620**, processor elements (PEs) **640**, and shared context storage files **660**. The following detailed description starts with the logical resource drivers since the TOLL software output is loaded into this hardware.

a. Logical Resource Drivers (LRDs)

The details of a particular logical resource driver (LRD) is set forth in FIG. **15**. As shown in FIG. **6**, each logical resource driver **620** is interconnected to the LRD-memory network **630** on one side and to the processor elements **640** through the PE-LRD network **650** on the other side. If the present invention were a SIMD machine, then only one LRD is provided and only one context file is provided. For MIMD capabilities, one

LRD and one context file is provided for each user so that, in the embodiment illustrated in FIG. 6, up to "n" users can be accommodated.

The logical resource driver **620** is composed of a data cache section **1500** and an instruction selection section **1510**. In the instruction selection section, the following components are interconnected. An instruction cache address translation unit (ATU) **1512** is interconnected to the LRD-memory network **630** over a bus **1514**. The instruction cache ATU **1512** is further interconnected over a bus **1516** to an instruction cache control circuit **1518**. The instruction cache control circuit **1518** is interconnected over lines **1520** to a series of cache partitions **1522***a*, **1522***b*, **1522***c*, and **1522***d*. Each of the cache partitions is respectively connected over busses **1524***a*, **1524***b*, **1524***c*, and **1524***d* to the LRD-memory network **630**. Each cache partition circuit is further interconnected over lines **1536***a*, **1536***b*, **1536***c*, and **1536***d* to a processor instruction queue (PIQ) bus interface unit **1544**. The PIQ bus interface unit **1544** is connected over lines **1546** to a branch execution unit (BEU) **1548** which in turn is connected over lines **1550** to the PE-context file network **670**. The PIQ bus interface unit **1544** is further connected over lines **1552***a*, **1552***b*, **1552***c*, and **1552***d* to a processor instruction queue (PIQ) buffer unit **1560** which in turn is connected over lines **1562***a*, **1562***b*, **1562***c*, and **1562***d* to a processor instruction queue (PIQ) processor assignment circuit **1570**. The PIQ processor assignment circuit **1570** is in turn connected over lines **1572***a*, **1572***b*, **1572***c*, and **1572***d* to the PE-LRD network **650** and hence to the processor elements **640**.

On the data cache portion **1500**, a data cache ATU **1580** is interconnected over bus **1582** to the LRD-memory network **630** and is further interconnected over bus **1584** to a data cache control circuit **1586** and over lines **1588** to a data cache interconnection network **1590**. The data cache control **1586** is also interconnected to data cache partition circuits **1592***a*, **1592***b*, **1592***c* and **1592***d* over lines **1593**. The data cache partition circuits, in turn, are interconnected over lines **1594***a*, **1594***b*, **1594***c*, and **1594***d* to the LRD-memory network **630**. Furthermore, the data cache partition circuits **1592** are interconnected over lines **1596***a*, **1596***b*, **1596***c*, and **1596***d* to the data cache interconnection network **1590**. Finally, the data cache interconnection network **1590** is interconnected over lines **1598***a*, **1598***b*, **1598***c*, and **1598***d* to the PE-LRD network **650** and hence to the processor elements **640**.

In operation, each logical resource driver (LRD) **620** has two sections, the data cache portion **1500** and the instruction selection portion **1510**. The data cache portion **1500** acts as a high speed data buffer between the processor elements **640** and memory **610**. Note that due to the number of memory requests that must be satisfied per unit time, the data cache **1500** is interleaved. All data requests made to memory by a processor element **640** are issued on the data cache interconnection network **1590** and intercepted by the data cache **1592**. The requests are routed to the appropriate data cache **1592** by the data cache interconnection network **1590** using the context identifier that is part of the dynamic SCSM information attached by the LRD to each instruction that is executed by the processors. The address of the desired datum determines in which cache partition the datum resides. If the requested datum is present (that is, a data cache hit occurs), the datum is sent back to the requesting processor element **640**.

If the requested datum is not present in data cache, the address delivered to the cache **1592** is sent to the data cache ATU **1580** to be translated into a system address. The system address is then issued to memory. In response, a block of data from memory (a cache line or block) is delivered into the cache partition circuits **1592** under control of data cache control **1586**. The requested data, that is resident in this cache block, is then sent through the data cache interconnection network **1590** to the requesting processor element **640**. It is to be expressly understood that this is only one possible design. The data cache portion is of conventional design and many possible implementations are realizable to one skilled in the art. As the data cache is of standard functionality and design, it will not be discussed further.

The instruction selection portion **1510** of the LRD has three major functions; instruction caching, instruction queueing and branch execution. The system function of the instruction cache portion of selection portion **1510** is typical of any instruction caching mechanism. It acts as a high speed instruction buffer between the processors and memory. However, the current invention presents methods and an apparatus configuration for realizing this function that are unique.

One purpose of the instruction portion **1510** is to receive execution sets from memory, place the sets into the caches **1522** and furnish the instructions within the sets, on an as needed basis, to the processor elements **640**. As the system contains multiple, generally independent, processor elements **640**, requests to the instruction cache are for a group of concurrently executable instructions. Again, due to the number of requests that must be satisfied per unit time, the instruction cache is interleaved. The group size ranges from none to the number of processors available to the users. The groups are termed packets, although this does not necessarily imply that the instructions are stored in a contiguous manner. Instructions are fetched from the cache on the basis of their instruction firing time (IFT). The next instruction firing time register contains the firing time of the next packet of instructions to be fetched. This register may be loaded by the branch execution unit **1548** of the LRD as well as incremented by the cache control unit **1518** when an instruction fetch has been completed.

The next IFT register (NIFTR) is a storage register that is accessible from the context control unit **1518** and the branch execution unit **1548**. Due to its simple functionality, it is not explicitly shown. Technically, it is a part of the instruction cache control unit **1518**, and is further buried in the control unit **1660** (FIG. **16**). The key point here is that the NIFTR is merely a storage register which can be incremented or loaded.

The instruction cache selection portion **1510** receives the instructions of an execution set from memory over bus **1524** and, in a round robin manner, places instructions word into each cache partitions, **1522***a*, **1522***b*, **1522***c* and **1522***d*. In other words, the instructions in the execution set are directed so that the first instruction is delivered to cache partition **1522***a*, the second instruction to cache partition **1522***b*, the third instruction to cache partition **1522***c*, and the fourth instruction to cache partition **1522***d*. The fifth instruction is then directed to cache partition **1522***a*, and so on until all of the instructions in the execution set are delivered into the cache partition circuits.

All the data delivered to the cache partitions are not necessarily stored in the cache. As will be discussed, the execution set header and trailer may not be stored. Each

cache partition attaches a unique identifier (termed a tag) to all the information that is to be stored in that cache partition. The identifier is used to verify that information obtained from the cache is indeed the information desired.

When a packet of instructions is requested, each cache partition determines if the partition contains an instruction that is a member of the requested packet. If none of the partitions contain an instruction that is a member of the requested packet (that is, a miss occurs), the execution set that contains the requested packet is requested from memory in a manner analogous to a data cache miss.

If a hit occurs (that is, at least one of the partitions **1522** contains an instruction from the requested packet), the partition(s) attach any appropriate dynamic SCSM information to the instruction(s). The dynamic SCSM information, which can be attached to each instruction, is important for multi-user applications. The dynamically attached SCSM information identifies the context file (see FIG. **6**) assigned to a given user. Hence, under the teachings of the present invention, the system **600** is capable of delay free switching among many user context files without requiring a master processor or access to memory.

The instruction(s) are then delivered to the PIQ bus interface unit **1544** of the LRD **620** where it is routed to the appropriate PIQ buffers **1560** according to the logical processor number (LPN) contained in the extended intelligence that the TOLL software, in the illustrated embodiment, has attached to the instruction. The instructions in the PIQ buffer unit **1560** are buffered for assignment to the actual processor elements **640**. The processor assignment is performed by the PIQ processor assignment unit **1570**. The assignment of the physical processor elements is performed on the basis of the number of processor elements currently available and the number of instructions that are available to be assigned. These numbers are dynamic. The selection process is set forth below.

The details of the instruction cache control **1518** and of each cache partition **1522** of FIG. **15** are set forth in FIG. **16**. In each cache partition circuit **1522**, five circuits are utilized. The first circuit is the header route circuit **1600** which routes an individual word in the header of the execution set over a path **1520***b* to the instruction cache context control unit **1660**. The control of the header route circuit **1600** is effected over path **1520***a* by the header path select circuit **1602**. The header path select circuit **1602** based upon the address received over lines **1520***b* from the control unit **1660** selectively activates the required number of header routers **1600** in the cache partitions. For example, if the execution set has two header words, only the first two header route circuits **1600** are activated by the header path select circuit **1602** and therefore two words of header information are delivered over bus **1520***b* to the control unit **1660** from the two activated header route circuits **1600** of cache partition circuits **1522***a* and **1522***b* (not shown). As mentioned, successive words in the execution set are delivered to successive cache partition circuits **1522**.

For example, assume that the data of Table 1 represents an entire execution set and that appropriate header words appear at the beginning of the execution set. The instructions with the earliest instruction firing times (IFTs) are listed first and for a given IFT, those instructions with the lowest logical processor number are listed first. The table reads:

TABLE 14

| Header Word 1<br>Header Word 2 | | |
|---|---|---|
| I0 | (T16) | (PEO) |
| I1 | (T16) | (PE1) |
| I4 | (T16) | (PE2) |
| I2 | (T17) | (PE0) |
| I5 | (T17) | (PE1) |
| I3 | (T18) | (PE0) |

Hence, the example of Table 1 (that is, the matrix multiply inner loop), now has associated with it two header words and the extended information defining the firing time (IFT) and the logical processor number (LPN). As shown in Table 14, the instructions were reordered by the TOLL software according to the firing times. Hence, as the execution set shown in Table 14 is delivered through the LRD-memory network **630** from memory, the first word (Header Word **1**) is routed by partition CACHE**0** to the control unit **1660**. The second word (Header Word **2**) is routed by partition CACHE**1** (FIG. **15**) to the control unit **1660**. Instruction **I0** is delivered to partition CACHE**2**, instruction **I1** to partition CACHE**3**, instruction **I2** to partition CACHE**0**, and so forth. As a result, the cache partitions **1522** now contain the instructions as shown in Table 15:

TABLE 15

| Cache0 | Cache1 | Cache2 | Cache3 |
|---|---|---|---|
| | | I0 | I1 |
| I4 | I2 | I5 | I3 |

It is important to clarify that the above example has only one basic block in the execution set (that is, it is a simplistic example). In actuality, an execution set would have a number of basic blocks.

The instructions are then delivered for storage into a cache random access memory (RAM) **1610** resident in each cache partition. Each instruction is delivered from the header router **1600** over a bus **1602** to the tag attacher circuit **1604** and then over a line **1606** into the RAM **1610**. The tag attacher circuit **1604** is under control of a tag generation circuit **1612** and is interconnected therewith over a line **1520***c*. Cache RAM **1610** could be a conventional cache high speed RAM as found in conventional superminicomputers.

The tag generation circuit **1612** provides a unique identification code (ID) for attachment to each instruction before storage of that instruction in the designated RAM **1610**. The assigning of process identification tags to instructions stored in cache circuits is conventional and is done to prevent aliasing of the instructions. "Cache Memories" by Alan J. Smith, ACM Computing Surveys, Vol. 14, September, 1982. The tag comprises a sufficient amount of information to uniquely identify it from each other instruction and user. The illustrated instructions already include the IFT and LPN, so that subsequently, when instructions are retrieved for execution, they can be fetched based on their firing times. As shown in Table 16, below, each instruction containing the extended information and the hardware tag is stored, as shown, for the above example:

TABLE 16

| | |
|---|---|
| CACHE0: | I4(T16)(PE2)(ID2) |
| CACHE1: | I2(T17)(PE0)(ID3) |
| CACHE2: | I0(T16)(PE0)(ID0) |
| | I5(T17)(PE1)(ID4) |
| CACHE3: | I1(T16)(PE1)(ID1) |

TABLE 16-continued

| I3(T18)(PE0)(ID5) |
|---|

As stated previously, the purpose of the cache partition circuits **1522** is to provide a high speed buffer between the slow main memory **610** and the fast processor elements **640**. Typically, the cache RAM **1610** is a high speed memory capable of being quickly accessed. If the RAM **1610** were a true associative memory, as can be witnessed in Table 16, each RAM **1610** could be addressed based upon instruction firing times (IFTs). At the present time, such associative memories are not economically justifiable and an IFT to cache address translation circuit **1620** must be utilized. Such a circuit is conventional in design and controls the addressing of each RAM **1610** over a bus **1520***d*. The purpose of circuit **1620** is to generate the RAM address of the desired instructions given the instruction firing time. Hence, for instruction firing time **T16**, CACHE0, CACHE2, and CACHE3, as seen in Table 16, would produce instructions **I4**, **I0**, and **I1** respectively.

When the cache RAMs **1610** are addressed, those instructions associated with a specific firing time are delivered over lines **1624** into a tag compare and privilege check circuit **1630**. The purpose f the tag compare and privilege check circuit **1630** is to compare the hardware tags (ID) to generated tags to verify that the proper instruction has been delivered. The reference tag is generated through a second tag generation circuit **1632** which is interconnected to the tag compare and privilege check circuit **1630** over a line **1520***e*. A privilege check is also performed on the delivered instruction to verify that the operation requested by the instruction is permitted given the privilege status of the process (e.g., system program, application program, etc.). This is a conventional check performed by computer processors which support multiple levels of processing states. A hit/miss circuit **1640** determines which RAMs **1610** have delivered the proper instructions to the PIQ bus interface unit **1544** in response to a specific instruction fetch request.

For example, and with reference back to Table 16, if the RAMs **1610** are addressed by circuit **1620** for instruction firing time **T16**, CACHE0, CACHE2, and CACHE3 would respond with instructions thereby comprising a hit indication on those cache partitions. Cache **1** would not respond and that would constitute a miss indication and this would be determined by circuit **1640** over line **1520***g*. Thus, each instruction, for instruction firing time **T16**, is delivered over bus **1632** into the SCSM attacher **1650** wherein dynamic SCSM information, if any, is added to the instruction by an SCSM attacher hardware **1650**. For example, hardware **1650** can replace the static SCSM procedural level information (which is a relative value) with the actual procedural level values. The actual values are generated from a procedural level counter data and the static SCSM information.

When all of the instructions associated with an individual firing time have been read from the RAM **1610**, the hit and miss circuit **1640** over lines **1646** informs the instruction cache control unit **1660** of this information. The instruction cache context control unit **1660** contains the next instruction firing time register, a part of the instruction cache control **1518** which increments the instruction firing time to the next value. Hence, in the example, upon the completion of reading all instructions associated with instruction firing time **T16**, the instruction cache context control unit **1660** increments to the next firing time, **T17**, and delivers this information over lines **1664** to an access resolution circuit **1670**, and over lines **1520***f* to the tag compare and privilege check circuit **1630**. Also note that there may be firing times which have no valid instructions, possibly due to operational dependencies detected by the TOLL software. In this case, no instructions would be fetched from the cache and transmitted to the PIQ interface.

The access resolution circuit **1670** coordinates which circuitry has access to the instruction cache RAMs **1610**. Typically, these RAMs can satisfy only a single request at each clock cycle. Since there could be two requests to the RAMs at one time, an arbitration method must be implemented to determine which circuitry obtains access. This is a conventional issue in the design of cache memory, and the access resolution circuit resolves the priority question as is well known in the field.

The present invention can and preferably does support several users simultaneously in both time and space. In previous prior art approaches (CDC, IBM, etc.), multi-user support was accomplished solely by timesharing the processor(s). In other words, the processors were shared in time. In this system, multi-user support is accomplished (in space) by assigning an LRD to each user that is given time on the processor elements. Thus, there is a spatial aspect to the sharing of the processor elements. The operating system of the machine deals with those users assigned to the same LRD in a timeshared manner, thereby adding the temporal dimension to the sharing of the processors.

Multi-user support is accomplished by the multiple LRDs, the use of plural processor elements, and the multiple context files **660** supporting the register files and condition code storage. As several users may be executing in the processor elements at the same time, additional pieces of information must be attached to each instruction prior to its execution to uniquely identify the instruction source and any resources that it may use. For example, a register identifier must contain the absolute value of the subroutine procedural level and the context identifier as well as the actual register number. Memory addresses must also contain the LRD identifier from which the instruction was issued to be properly routed through the LRD-Memory interconnection network to the appropriate data cache.

The additional and required information comprises two components, a static and a dynamic component; and the information is termed "shared context storage mapping" (SCSM). The static information results from the compiler output and the TOLL software gleans the information from the compiler generated instruction stream and attaches the register information to the instruction prior to its being received by an LRD.

The dynamic information is hardware attached to the instruction by the LRD prior to its issuance to the processors. This information is composed of the context/LRD identifier corresponding to the LRD issuing the instruction, the absolute value of the current procedural level of the instruction, the process identifier of the current instruction stream, and preferably the instruction status information that would normally be contained in the processors of a system having processors that are not context free. This later information would be composed of error masks, floating point format modes, rounding modes, and so on.

In the operation of the circuitry in FIG. **16**, one or more execution sets are delivered into the instruction cache circuitry. The header information for each set is delivered to one or more successive cache partitions and is routed to the context control unit **1660**. The instructions in the execution set are then individually, on a round robin basis, routed to each successive cache partition unit **1522**. A hardware identification tag is attached to each instruction and the instruction is then stored in RAM **1610**. As previously discussed, each execution set is of sufficient length to minimize instruction cache defaults and the RAM **1610** is of sufficient size to store the execution sets. When the processor elements require the instructions, the number and cache locations of the valid instructions matching the appropriate IFTs are determined. The instructions stored in the RAM's **1610** are read out; the identification tags are verified; and the privilege status checked. The instructions are then delivered to the PIQ bus interface unit **1544**. Each instruction that is delivered to the PIQ bus interface unit **1544**, as is set forth in Table 17, includes the identification tag (ID) and the hardware added SCSM information.

TABLE 17

| | |
|---|---|
| CACHE0: | I4(T16)(PE2)(ID2)(SCSM0) |
| CACHE1: | I2(T17)(PE0)(ID3)(SCSM1) |
| CACHE2: | I0(T16)(PE0)(ID0)(SCSM2) |
| | I5(T17)(PE1)(ID4)(SCSM3) |
| CACHE3: | I1(T16)(PE1)(ID1)(SCSM4) |
| | I3(T18)(PE0)(ID5)(SCSM5) |

If an instruction is not stored in RAM **1610**, a cache miss occurs and a new execution set containing the instruction is read from main memory over lines **1523**.

In FIG. **17**, the details of the PIQ bus interface unit **1544** and the PIQ buffer unit **1560** are set forth. Referring to FIG. **17**, the PIQ bus interface unit **1544** receives instructions as set forth in Table 17, above, over lines **1536**. A search tag hardware **1702** has access to the value of the present instruction firing time over lines **1549** and searches the cache memories **1522** to determine the address(es) of those registers containing instructions having the correct firing times. The search tag hardware **1702** then makes available to the instruction cache control circuitry **1518** the addresses of those memory locations for determination by the instruction cache control of which instructions to next select for delivery to the PIQ bus interface **1544**.

These instructions access, in parallel, a two-dimensional array of bus interface units (BIU's) **1700**. The bus interface units **1700** are interconnected in a full access non-blocking network by means of connections **1710** and **1720**, and connect over lines **1552** to the PIQ buffer unit **1560**. Each bus interface unit (BIU) **1700** is a conventional address comparison circuit composed of: TI 74L85 4 bit magnitude comparators, Texas Instruments Company, P.O. Box 225012, Dallas, Tex. 75265. In the matrix multiply example, for instruction firing time **T16**, CACHE0 contains instruction **I4** and CACHE3 (corresponding to CACHE n in FIG. **17**) contains instruction **I1**. The logical processor number assigned to instruction **I4** is PE2. The logical processor number PE2 activates a select (SEL) signal of the bus interface unit **1700** for processor instruction queue **2** (this is the BIU3 corresponding to the CACHE0 unit containing the instruction). In this example, only that BIU3 is activated and the remaining bus interface units **1700** for that BIU3 row and column are not activated. Likewise, for CACHE3 (CACHE n in FIG. **17**), the corresponding BIU2 is activated for processor instruction QUEUE **1**.

The PIQ buffer unit **1560** is comprised of a number of processor instruction queues **1730** which store the instructions received from the PIQ bus interface unit **1544** in a first in-first out (FIFO) fashion as shown in Table 18:

TABLE 18

| PIQ0 | PIQ1 | PIQ2 | PIQ3 |
|---|---|---|---|
| I0 | I1 | I4 | — |
| I2 | — | — | — |
| I3 | — | — | — |

In addition to performing instruction queueing functions, the PIQ's **1730** also keep track of the execution status of each instruction that is issued to the processor elements **640**. In an ideal system, instructions could be issued to the processor elements every clock cycle without worrying about whether or not the instructions have finished execution. However, the processor elements **640** in the system may not be able to complete an instruction every clock cycle due to the occurrence of exceptional conditions, such as a data cache miss and so on. As a result, each PIQ **1730** tracks all instructions that it has issued to the processor elements **640** that are still in execution. The primary result of this tracking is that the PIQ's **1730** perform the instruction clocking function for the LRD **620**. In other words, the PIQ's **1730** determine when the next firing time register can be updated when executing straightline code. This in turn begins a new instruction fetch cycle.

Instruction clocking is accomplished by having each PIQ **1730** form an instruction done signal that specifies that the instruction(s) issued by a given PIQ either have executed or, in the case of pipelined PE's, have proceeded to the next stage. This is then combined with all other PIQ instruction done signals from this LRD and is used to gate the increment signal that increments the next firing time register. The "done" signals are delivered over lines **1564** to the instruction cache control **1518**.

Referring to FIG. **18**, the PIQ processor assignment circuit **1570** contains a two dimensional array of network interface units (NIU's) **1800** interconnected as a full access switch to the PE-LRD network **650** and then to the various processor elements **640**. Each network interface unit (NIU) **1800** is comprised of the same circuitry as the bus interface units (BIU) **1700** of FIG. **17**. In normal operation, the processor instruction queue #0 (PIQ0) can directly access processor element **0** by activating the NIU0 associated with the column corresponding to queue #0, the remaining network interface units NIU0, NIU1, NIU2, NIU3 of the PIQ processor alignment circuit for that column and row being deactivated. Likewise, processor instruction queue #3 (PIQ3) normally accesses processor element **3** by activating the NIU3 of the column associated with queue #3, the remaining NIU0, NIU1, NIU2, and NIU3 of that column and row being deactivated. The activation of the network interface units **1800** is under the control of an instruction select and assignment unit **1810**. Unit **1810** receives signals from the PIQ's **1730** within the LRD that the unit **1810** is a member of over lines **1811**, from all other units **1810** (of other LRD's) over lines **1813**, and from the processor elements **640** through the network **650**. Each PIQ **1730** furnishes the unit **1810** with a signal that corresponds to "I have an instruction that is

ready to be assigned to a processor." The other PIQ buffer units furnish this unit **1810** and every other unit **1810** with a signal that corresponds to "My PIQ **1730** (#x) has an instruction ready to be assigned to a processor." Finally, the processor elements furnish each unit **1810** in the system with a signal that corresponds to "I can accept a new instruction."

The unit **1810** on an LRD transmits signals to the PIQs **1730** of its LRD over lines **1811**, to the network interface units **1800** of its LRD over lines **1860** and to the other units **1810** of the other LRDs in the system over lines **1813**. The unit **1810** transmits a signal to each PIQ **1730** that corresponds to "Gate your instruction onto the PE-LRD interface bus (**650**)." The unit transmits a select signal to the network interface units **1800**. Finally, the unit **1810** transmits a signal that corresponds to "I have used processor element #x" to each other unit **1810** in the system for each processor which it is using.

In addition, each unit **1810** in each LRD has associated with it a priority that corresponds to the priority of the LRD. This is used to order the LRDs into an ascending order from zero to the number of LRDs in the system. The method used for assigning the processor elements is as follows. Given that the LRDs are ordered, many allocation schemes are possible (e.g., round robin, first come first served, time slice, etc.). However, these are implementation details and do not impact the functionality of this unit under the teachings of the present invention.

Consider the LRD with the current highest priority. This LRD gets all the processor elements that it requires and assigns the instructions that are ready to be executed to the available processor elements. If the processor elements are context free, the processor elements can be assigned in any manner whatsoever. Typically, however, assuming that all processors are functioning correctly, instructions from PIQ #**0** are routed to processor element #**0**, provided of course, processor element #**0** is available.

The unit **1810** in the highest priority LRD then transmits this information to all other units **1810** in the system. Any processors left open are then utilized by the next highest priority LRD with instructions that can be executed. This allocation continues until all processors have been assigned. Hence, processors may be assigned on a priority basis in a daisy chained manner.

If a particular processor element, for example, element **1** has failed, the instruction selective assignment unit **1810** can deactivate that processor element by deactivating all network instruction units NIU**1**. It can then, through hardware, reorder the processor elements so that, for example, processor element **2** receives all instructions logically assigned to processor element **1**, processor element **3** is now assigned to receive all instructions logically assigned to processor **2**, etc. Indeed, redundant processor elements and network interface units can be provided to the system to provide for a high degree of fault tolerance.

Clearly, this is but one possible implementation. Other methods are also realizable.

b. Branch Execution Unit (BEU)

Referring to FIG. **19**, the Branch Execution Unit (BEU) **1548** is the unit in the present invention responsible for the execution of all branch instructions which occur at the end of each basic block. There is, in the illustrated embodiment, one BEU **1548** for each supported context and so, with reference to FIG. **6**, "n" supported contexts require "n" BEU's. The illustrated embodiment uses one BEU for each supported context because each BEU **1548** is of simple design and, therefore, the cost of sharing a BEU between plural contexts would be more expensive than allowing each context to have its own BEU.

The BEU **1548** executes branches in a conventional manner with the exception that the branch instructions are executed outside the PE's **640**. The BEU **1548** evaluates the branch condition and, when the target address is selected, generates and places this address directly into the next instruction fetch register. The target address generation is conventional for unconditional and conditional branches that are not procedure calls or returns. The target address can be (a) taken directly from the instruction, (b) an offset from the current contents of the next instruction fetch register, or (c) an offset of a general purpose register of the context register file.

A return branch from a subroutine is handled in a slightly different fashion. To understand the subroutine return branch, discussion of the subroutine call branch is required. When the branch is executed, a return address is created and stored. The return address is normally the address of the instruction following the subroutine call. The return address can be stored in a stack in memory or in other storage local to the branch execution unit. In addition, the execution of the subroutine call increments the procedural level counter.

The return from a subroutine branch is also an unconditional branch. However, rather than containing the target address within the instruction, this type of branch reads the previously stored return address from storage, decrements the procedural level counter, and loads the next instruction fetch register with the return address. The remainder of the disclosure discusses the evaluation and execution of conditional branches. It should be noted the that techniques described also apply to unconditional branches, since these are, in effect, conditional branches in which the condition is always satisfied. Further, these same techniques also apply to the subroutine call and return branches, which perform the additional functions described above.

To speed up conditional branches, the determination of whether a conditional branch is taken or not, depends solely on the analysis of the appropriate set of condition codes. Under the teachings of the present invention, no evaluation of data is performed other than to manipulate the condition codes appropriately. In addition, an instruction, which generates a condition code that a branch will use, can transmit the code to BEU **1548** as well as to the condition code storage. This eliminates the conventional extra waiting time required for the code to become valid in the condition code storage prior to a BEU being able to fetch it.

The present invention also makes extensive use of delayed branching to guarantee program correctness. When a branch has executed and its effects are being propagated in the system, all instructions that are within the procedural domain of the branch must either have been executed or be in the process of being executed, as discussed in connection with the example of Table 6. In other words, changing the next-instruction pointer (in response to the branch) takes place after the current firing time has been updated to point to the firing time that follows the last (temporally executed) instruction of the branch. Hence, in the example of Table 6, instruction **I5** at firing time **T17** is delayed until the completion

of T18 which is the last firing time for this basic block. The instruction time for the next basic block is then T19.

The functionality of the BEU **1548** can be described as a four-state state machine:

Stage 1: Instruction decode
- Operation decode
- Delay field decode
- Condition code access decode

Stage 2: Condition code fetch/receive

Stage 3: Branch operation evaluation

Stage 4: Next instruction fetch location and firing time update

Along with determining the operation to be performed, the first stage also determines how long fetching can continue to take place after receipt of the branch by the BEU, and how the BEU is to access the condition codes for a conditional branch, that is, are they received or fetched.

Referring to FIG. **19**, the branch instruction is delivered over bus **1546** from the PIQ bus interface unit **1544** into the instruction register **1900** of the BEU **1548**. The fields of the instruction register **1900** are designated as: FETCH/ENABLE, CONDITION CODE ADDRESS, OP CODE, DELAY FIELD, and TARGET ADDRESS. The instruction register **1900** is connected over lines **1910***a* and **1910***b* to a condition code access unit **1920**, over lines **1910***c* to an evaluation unit **1930**, over lines **1910***d* to a delay unit **1940**, and over lines **1910***e* to a next instruction interface **1950**.

Once an instruction has been issued to BEU **1548** from the PIQ bus interface **1544**, instruction fetching must be held up until the value in the delay field has been determined. This value is measured relative to the receipt of the branch by the BEU, that is stage **1**. If there are no instructions that may be overlapped with this branch, this field value is zero. In this case, instruction fetching is held up until the outcome of the branch has been determined. If this field is non-zero, instruction fetching may continue for a number of firing times given by the value in this field.

The condition code access unit **1920** is connected to the register file - PE network **670** over lines **1550** and to the evaluation unit **1930** over lines **1922**. During stage **2** operation, the condition code access decode unit **1920** determines whether or not the condition codes must be fetched by the instruction, or whether the instruction that determines the branch condition delivers them. As there is only one instruction per basic block that will determine the conditional branch, there will never be more than one condition code received by the BEU for a basic block. As a result, the actual timing of when the condition code is received is not important. If it comes earlier than the branch, no other codes will be received prior to the execution of the branch. If it comes later, the branch will be waiting and the codes received will always be the right ones. Note that the condition code for the basic block can include plural codes received at the same or different times by the BEU.

The evaluation unit **1930** is connected to the next instruction interface **1950** over lines **1932**. The next instruction interface **1950** is connected to the instruction cache control circuit **1518** over lines **1549** and to the delay unit **1940** over lines **1942**; and the delay unit **1940** is also connected to the instruction cache control unit **1518** over lines **1549**.

During the evaluation stage of operation, the condition codes are combined according to a Boolean function that represents the condition being tested. In the final stage of operation, either fetching of the sequential instruction stream continues, if a conditional branch is not taken, or the next instruction pointer is loaded, if the branch is taken.

The impact of a branch in the instruction stream can be described as follows. Instructions, as discussed, are sent to their respective PIQ's **1730** by analysis of the resident logical processor number (LPN). Instruction fetching can be continued until a branch is encountered, that is, until an instruction is delivered to the instruction register **1900** of the BEU **1548**. At this point, in a conventional system without delayed branching, fetching would be stopped until the resolution of the branch instruction is complete. See, for example, "Branch Prediction Strategies and Branch Target Buffer Design", J. F. K. Lee & A. J. Smith, IEEE Computer Magazine, January, 1984.

In the present system, which includes delayed branching, instructions must continue to be fetched until the next instruction fetched is the last instruction of the basic block to be executed. The time that the branch is executed is then the last time that fetching of an instruction can take place without a possibility of modifying the next instruction address. Thus, the difference between when the branch is fetched and when the effects of the branch are actually felt corresponds to the number of additional firing time cycles during which fetching can be continued.

The impact of this delay is that the BEU **1548** must have access to the next instruction firing time register of the cache controller **1518**. Further, the BEU **1548** can control the initiation or disabling of the instruction fetch process performed by the instruction cache control unit **1518**. These tasks are accomplished by signals over bus **1549**.

In operation the branch execution unit (BEU) **1548** functions as follows. The branch instruction, such as instruction **I5** in the example above, is loaded into the instruction register **1900** from the PIQ bus interface unit **1544**. The contents of the instruction register then control the further operation of BEU **1548**. The FETCH-ENABLE field indicates whether or not the condition code access unit **1920** should retrieve the condition code located at the address stored in the CC-ADX field (called FETCH) or whether the condition code will be delivered by the generating instruction.

If a FETCH is requested, the unit **1920** accesses the register file-PE network **670** (see FIG. **6**) to access the condition code storage **2000** which is shown in FIG. **20**. Referring to FIG. **20**, the condition code storage **2000**, for each context file, is shown in the generalized case. A set of registers CCxy are provided for storing condition codes for procedural level y. Hence, the condition code storage **2000** is accessed and addressed by the unit **1920** to retrieve, pursuant to a FETCH request, the necessary condition code. The actual condition code and an indication that the condition code is received by the unit **1920** is delivered over lines **1922** to the evaluation unit **1930**. The OPCODE field, delivered to the evaluation unit **1930**, in conjunction with the received condition code, functions to deliver a branch taken signal over line **1932** to the next instruction interface **1950**. The evaluation unit **1930** is comprised of standard gate arrays such as those from LSI Logic Corporation, 1551 McCarthy Blvd., Milpitas, Calif. 95035.

The evaluation unit **1930** accepts the condition code set that determines whether or not the conditional

branch is taken, and under control of the OPCODE field, combines the set in a Boolean function to generate the conditional branch taken signal.

The next instruction interface **1950** receives the branch target address from the TARGET-ADX field of the instruction register **1900** and the branch taken signal over line **1932**. However, the interface **1950** cannot operate until an enable signal is received from the delay unit **1940** over lines **1942**.

The delay unit **1940** determines the amount of time that instruction fetching can be continued after the receipt of a branch instruction by the BEU. Previously, it has been described that when a branch instruction is received by the BEU, instruction fetching continues for one more cycle and then stops. The instruction fetched during this cycle is held up from passing through PIQ bus interface unit **1544** until the length of the delay field has been determined. For example, if the delay field is zero (implying that the branch is to be executed immediately), these instructions must still be withheld from the PIQ bus buffer unit until it is determined whether or not these are the right instructions to be fetched. If the delay field is non-zero, the instructions would be gated into the PIQ buffer unit as soon as the delay value was determined to be non-zero. The length of the delay is obtained from DELAY field of the instruction register **1900**. The delay unit receives the delay length from register **1900** and clock impulses from the context control **1518** over lines **1549**. The delay unit **1940** decrements the value of the delay at each clock pulse; and when fully decremented, the interface unit **1950** becomes enabled.

Hence, in the discussion of Table 6, instruction **I5** is assigned a firing time **T17** but is delayed until firing time **T18**. During the delay time, the interface **1950** signals the instruction cache control **1518** over line **1549** to continue to fetch instructions to finish the current basic block. When enabled, the interface unit **1950** delivers the next address (that is, the branch execution address) for the next basic block into the instruction cache control **1518** over lines **1549**.

In summary and for the example on Table 6, the branch instruction **I5** is loaded into the instruction register **1900** during time **T17**. However, a delay of one firing time (DELAY) is also loaded into the instruction register **1900** as the branch instruction cannot be executed until the last instruction **I3** is processed during time **T18**. Hence, even though the instruction **I5** is loaded in register **1900**, the branch address for the next basic block, which is contained in the TARGET ADDRESS, does not become effective until the completion of time **T18**. In the meantime, the next instruction interface **1950** issues instructions to the cache control **1518** to continue processing the stream of instructions in the basic block. Upon the expiration of the delay, the interface **1950** is enabled, and the branch is executed by delivering the address of the next basic block to the instruction cache control **1518**.

Note that the delay field is used to guarantee the execution of all instructions in the basic block governed by this branch in single cycle context free PE's. A small complexity is encountered when the PE's are pipelined. In this case, there exist data dependencies between the instructions from the basic block just executed, and the instructions from the basic block to be executed. The TOLL software can analyze these dependencies when the next basic block is only targeted by the branch from this basic block. If the next basic block is targeted by more than one branch, the TOLL software cannot resolve the various branch possibilities and lets the pipelines drain, so that no data dependencies are violated. One mechanism for allowing the pipelines to drain is to insert NO-OP (no operation) instructions into the instruction stream. An alternate method provides an extra field in the branch instruction which inhibits the delivery of new instructions to the processor elements for a time determined by the data in the extra field.

c. Processor Elements (PE)

So far in the discussions pertaining to the matrix multiply example, a single cycle processor element has been assumed. In other words, an instruction is issued to the processor element and the processor element completely executes the instruction before proceeding to the next instruction. However, greater performance can be obtained by employing pipelined processor elements. Accordingly, the tasks performed by the TOLL software change slightly. In particular, the assignment of the processor elements is more complex than is shown in the previous example; and the hazards that characterize a pipeline processor must be handled by the TOLL software. The hazards that are present in any pipelined processor manifest themselves as a more sophisticated set of data dependencies. This can be encoded into the TOLL software by one practiced in the art. See for example, T. K. R. Gross, Stanford University, 1983, "Code Optimization of Pipeline Constraints", Doctorate Dissertation Thesis.

The assignment of the processors is dependent on the implementation of the pipelines and again, can be performed by one practiced in the art. A key parameter is determining how data is exchanged between the pipelines. For example, assume that each pipeline contains feedback paths between its stages. In addition, assume that the pipelines can exchange results only through the register sets **660**. Instructions would be assigned to the pipelines by determining sets of dependent instructions that are contained in the instruction stream and then assigning each specific set to a specific pipeline. This minimizes the amount of communication that must take place between the pipelines (via the register set), and hence speeds up the execution time of the program. The use of the logical processor number guarantees that the instructions will execute on the same pipeline.

Alternatively, if there are paths available to exchange data between the pipelines, dependent instructions may be distributed across several pipeline processors instead of being assigned to a single pipeline. Again, the use of multiple pipelines and the interconnection network between them that allows the sharing of intermediate results manifests itself as a more sophisticated set of data dependencies imposed on the instruction stream. Clearly, the extension of the teachings of this invention to a pipelined system is within the skill of one practiced in the art.

Importantly, the additional data (chaining) paths do not change the fundamental context free nature of the processor elements of the present invention. That is, at any given time (for example, the completion of any given instruction cycle), the entire process state associated with a given program (that is, context) is captured completely external to the processor elements. Data chaining results merely in a transitory replication of some of the data generated within the processor elements during a specific instruction clock cycle.

Referring to FIG. **21**, a particular processor element **640** has a four-stage pipeline processor element. All